Feature Flags

Play+ Feature Flag Helper Introduction ■ In the Play+ ecosystem, we believe shipping code should be empowering, not risky. This helper is based on the concept of decoupling deployment from release , a modern development practice that allows us to merge and deploy features to production safely, while precisely controlling their visibility. Feature flags (or toggles) are essential for mitigating risk, enabling A/B testing, and personalizing user experiences at scale. This aligns directly with our Adaptive pillar, allowing the application to change dynamically based on user context or business needs. It also promotes an Intuitive development workflow by simplifying a complex topic and makes the product more Engaging by allowing for controlled experiments with new features. Package Info ■ The playfeature helper is provided through the @playplus/features package, which is included by default in the Golden Path. Description Package / Path Golden Path (Recommended) Pre-installed (/system/play.feature.ts) Uplift Path npm install @playplus/features Folder Reference ■ The feature flag helper and its configuration follow our standardized folder structure for core system logic. File / Directory Purpose & Guidelines system/play.feature.ts The core feature flag service. It abstracts the third-party provider and provides a consistent API. config/play.feature.config.json User-overridable configuration for the flag provider, client keys, and default values for offline development. Helper - Pillars Alignment ■ The playfeature helper is a key enabler of our core design pillars. Pillar How This Helper Aligns Adaptive Primary Pillar : Allows the application's UI and behavior to adapt in real-time based on user, region, or experimental group. Intuitive Abstracts the complexity of third-party SDKs into a simple, synchronous API, making it trivial for developers to use flags. Engaging Empowers teams to safely A/B test new, delightful features and interactions, gathering data to create more engaging experiences. Helper Overview ■ The playfeature helper is your application's runtime control panel. It provides a clean, synchronous, and framework-agnostic interface to your feature flagging provider (e.g., LaunchDarkly, Optimizely, Flagsmith). Its purpose is to abstract the plumbing of flag management, so developers can focus on building features, not on SDK integration. Behind the scenes, the helper automates the entire process: Initialization : On app startup, it connects to your flag provider using the configured client key. User Identification : It automatically identifies the current user to fetch targeted flags. Real-time Updates : It maintains a live connection, so any change made in your provider's dashboard is reflected in the app in real-time, without a page refresh. Offline Graceful Degradation : If the provider is unreachable, it falls back to default values defined in the configuration. This means a developer can simply ask playfeature.isEnabled('my-flag') and trust that the system is handling all the complex state management in the background. Config Options ■ Global configuration is managed in config/play.feature.config.json . Config Variable Default Value Description Recommended Value provider "local" The name of your feature flag provider (e.g., launchdarkly, flagsmith, optimizely). local uses bootstrap values. Your provider's name. clientKey null The client-side ID or key for your feature flag project. Your provider's client

key. bootstrap {} An object of default flag values to use during initial load or for offline development and testing. Define key flags for testing. Example play.feature.config.json: { "provider" : "launchdarkly" , "clientKey" : "sdk-12345-abcde" , "bootstrap" : { "feat-new-dashboard-2025-q3" : false , "enable-beta-analytics" : true } } Helper Methods ■ The helper provides a simple, consistent API for accessing flag values. Method Name What It Does Method Signature isEnabled Checks if a boolean feature flag is enabled. Returns defaultValue if the flag doesn't exist. isEnabled(key: string, defaultValue?: boolean): boolean getVariant Gets the string value for a multivariate or experiment flag. getVariant(key: string, defaultVariant?: string): string onUpdate Subscribes to all flag changes from the provider, allowing the UI to react in real-time. onUpdate(callback: () => void): void offUpdate Unsubscribes from flag changes to prevent memory leaks. offUpdate(callback: () => void): void Angular Integration ■ PlayFeatureService ■ Angular service wrapper that integrates with Play+ logging and provides component-specific feature flag utilities. import { PlayFeatureService } from '../services/playfeature.service' ; @ Component ({ ... }) export class MyComponent { constructor (private playFeature : PlayFeatureService) { } ngOnInit ( ) { if (this . playFeature . isEnabled ('feat-new-dashboard-2025-q3' , false)) { // Show new dashboard } } } PlayFeatureGuard ■ Route protection guard for feature flag-based access control. // app-routing.module.ts import { PlayFeatureGuard } from "./guards/play-feature.guard" ; const routes : Routes = [{ path : "next-gen-dashboard" , loadChildren : ( ) => import ("./next-gen/dashboard.module") . then ((m) => m . DashboardModule) , canActivate : [PlayFeatureGuard] , data : { featureFlag : "feat-next-gen-dashboard-enabled" , fallbackRoute : "/dashboard" , } , } ,] ; FeatureFlagDirective ■ Template conditional rendering based on feature flags. <!-- Boolean flags --> < div *playFeatureFlag = " ' feat-new-dashboard-2025-q3' " > < h2 > New Dashboard </ h2 > < p > This content is only visible when the flag is enabled. </ p > </ div > <!-- Variant flags --> < div *playFeatureFlag = " ' feat-experiment'; playFeatureFlagVariant: 'treatment' " > < h2 > Experimental Feature </ h2 > < p > This shows the treatment variant. </ p > </ div > FeatureFlagPipe ■ Template flag checks and variant access. <!-- In *ngIf --> < ng-container *ngIf = " ' feat-new-analytics' | playFeatureFlag " > < app-analytics-widget > </ app-analytics-widget > </ ng-container > <!-- Display flag status --> < p > New Dashboard: {{ 'feat-new-dashboard' | playFeatureFlag ? 'Enabled' : 'Disabled' }} </ p > <!-- Variant checks --> < p > Experiment Variant: {{ 'feat-experiment' | playFeatureVariant:'control' }} </ p > Usage Examples ■ React: The useFeature Hook ■ For a seamless experience in React, the starter kit provides a useFeature hook that automatically subscribes to live updates. // hooks/useFeature.ts import { useState , useEffect } from "react" ; import { playfeature } from "../system/play.feature" ; export function useFeature (key : string , defaultValue : boolean) : boolean { const [value , setValue] = useState (( ) => playfeature . isEnabled (key , defaultValue)) ; useEffect (( ) => { // A function to update state when flags change. const handleUpdate = ( ) => { setValue (playfeature . isEnabled (key , defaultValue)) ; } ; // Set initial value and subscribe to live updates. handleUpdate ( ) ; playfeature . onUpdate (handleUpdate) ; // Clean up the subscription on component unmount. return ( ) => { playfeature . offUpdate (handleUpdate) ; } ; } , [key , defaultValue]) ; return value ; } // In a component: // MainDashboard.tsx import { useFeature } from "../hooks/useFeature" ;

```jsx
function MainDashboard ( ) { const showNewAnalyticsCard = useFeature (
"feat-new-analytics-card" , false ) ; return ( < div > < UserInfoCard /> { showNewAnalyticsCard ? <
NewAnalyticsCard /> : < OldAnalyticsCard /> } </ div > ) ; }
```

Angular: The PlayFeatureGuard for Routes ■ For Angular, a CanActivate guard is the perfect way to toggle access to entire feature routes.

```ts
// guards/play-feature.guard.ts import { Injectable } from "@angular/core" ; import {
CanActivate , ActivatedRouteSnapshot , Router } from "@angular/router" ; import { playfeature }
from "@playplus/core" ; // Assuming helper is available @ Injectable ( { providedIn : "root" } )
export class PlayFeatureGuard implements CanActivate { constructor ( private router : Router ) { }
canActivate ( route : ActivatedRouteSnapshot ) : boolean { const flagName = route . data [
"featureFlag" ] ; if ( flagName && playfeature . isEnabled ( flagName ) ) { return true ; // Allow
access to the route } // If flag is off or doesn't exist, redirect away this . router . navigate ( [
"/dashboard" ] ) ; return false ; } } // app-routing.module.ts import { PlayFeatureGuard } from
"./guards/play-feature.guard" ; const routes : Routes = [ { path : "next-gen-dashboard" ,
loadChildren : ( ) => import ( "./next-gen/dashboard.module" ) . then ( ( m ) => m .
DashboardModule ) , canActivate : [ PlayFeatureGuard ] , data : { featureFlag :
"feat-next-gen-dashboard-enabled" } , } , ] ;
```

Service-based Flag Checks ■ ** Recommended: Use PlayFeatureService**

```ts
import { PlayFeatureService } from '../services/playfeature.service' ; @
Component ( { ... } ) export class MyComponent { constructor ( private playFeature :
PlayFeatureService ) { } ngOnInit ( ) { if ( this . playFeature . isEnabled (
'feat-new-dashboard-2025-q3' , false ) ) { // Show new dashboard } } }
```

Reactive Flag Observables ■ ** Recommended: Use reactive observables**

```ts
import { PlayFeatureService } from
'../services/playfeature.service' ; @ Component ( { ... } ) export class MyComponent implements
OnInit , OnDestroy { private destroy$ = new Subject < void > ( ) ; newDashboardEnabled = false ;
constructor ( private playFeature : PlayFeatureService ) { } ngOnInit ( ) { this . playFeature . flag$ (
'feat-new-dashboard-2025-q3' , false ) . pipe ( takeUntil ( this . destroy$ ) ) . subscribe ( enabled
=> { this . newDashboardEnabled = enabled ; } ) ; } ngOnDestroy ( ) { this . destroy$ . next ( ) ; this
. destroy$ . complete ( ) ; } }
```

Additional Info ■ Why We Created This Helper ■ Without a centralized helper, each developer would need to: Integrate and learn a complex, third-party feature flag SDK. Manually handle user identification for targeted rollouts. Build their own logic for real-time updates and subscriptions. Write boilerplate code to handle cases where the flag provider is offline. This is inefficient and error-prone. The playfeature helper abstracts all of this into a single, reliable system. It provides a consistent, tested interface so developers can add or check a feature flag in a single line of code, trusting that the underlying complexity is managed for them. Best Practices ■ Keep Flags Short-Lived : Flags are temporary. Create a process for cleaning them up after a feature is fully rolled out to avoid technical debt. Use Descriptive Names : A good name like feat-newDashboard-rollout-2025-q3 is self-documenting. It tells you the feature, its purpose, and its expected lifespan. Abstract Flag Checks : Don't sprinkle playfeature.isEnabled() calls directly in JSX. Encapsulate the logic in a variable, hook (useFeature), or function for better readability and maintenance. Always Provide a Default Value : This ensures your application behaves predictably if the flagging service is down or a flag is

deleted. Flag Naming Convention ■ ** Good Names:** feat-new-dashboard-2025-q3 enable-beta-analytics feat-secure-input-validation feat-performance-monitoring ** Bad Names:** newDashboard (no prefix) beta (too vague) flag1 (not descriptive) temp (temporary flags become permanent) Default Values ■ ** Always provide safe defaults:** // Good - explicit default this . playFeature . isEnabled ("feat-new-dashboard" , false) ; // Good - explicit default for variants this . playFeature . getVariant ("feat-experiment" , "control") ; ** Avoid implicit defaults:** // Bad - relies on system default this . playFeature . isEnabled ("feat-new-dashboard") ; Flag Abstraction ■ ** Abstract flag checks:** // Good - abstracted in component export class DashboardComponent { private readonly showNewDashboard = this . playFeature . isEnabled ("feat-new-dashboard-2025-q3" , false) ; ngOnInit ( ) { if (this . showNewDashboard) { this . loadNewDashboard ( ) ; } else { this . loadLegacyDashboard ( ) ; } } } ** Don't sprinkle flag checks everywhere:** // Bad - flag check in template logic template : ` <div *ngIf="playFeature.isEnabled('feat-new-dashboard')"> <!-- content --> </div> ` ; Flag Lifecycle Management ■ ** Plan flag cleanup:** // Document flag purpose and cleanup plan /** * Flag: feat-new-dashboard-2025-q3 * Purpose: Rollout new dashboard to 50% of users * Cleanup: Remove after Q3 2025 when 100% rollout is complete * Owner: team-core-ui */ Forbidden Patterns ■ 1. Direct playfeature Access ■ ** Never access playfeature directly:** // DON'T DO THIS import { playfeature } from "../../../system/play.feature" ; if (playfeature . isEnabled ("my-flag")) { // This violates Play+ guidelines } 2. Magic String Flag Names ■ ** Don't use magic strings:** // DON'T DO THIS if (this . playFeature . isEnabled ("new-dashboard")) { // Flag name not in config } 3. Inline Flag Checks in Templates ■ ** Don't put complex flag logic in templates:** <!-- DON'T DO THIS --> < div *ngIf = " playFeature.isEnabled('feat-new-dashboard') && user.hasPermission('admin') " > <!-- Complex logic in template --> </ div > 4. Flag Checks in Services ■ ** Don't check flags in business logic services:** // DON'T DO THIS @ Injectable ( ) export class UserService { getUsers ( ) { if (this . playFeature . isEnabled ("feat-new-api")) { return this . newApi . getUsers ( ) ; } return this . legacyApi . getUsers ( ) ; } } Required Patterns ■ 1. Use PlayFeatureService ■ ** Always use the Angular service:** import { PlayFeatureService } from '../services/playfeature.service' ; constructor (private playFeature : PlayFeatureService) { } 2. Provide Default Values ■ ** Always provide explicit defaults:** this . playFeature . isEnabled ("feat-new-dashboard" , false) ; this . playFeature . getVariant ("feat-experiment" , "control") ; 3. Use Reactive Patterns ■ ** Use observables for reactive UI:** this . playFeature . flag$ ("feat-new-dashboard" , false) . pipe (takeUntil (this . destroy$)) . subscribe ((enabled) => { this . showNewDashboard = enabled ; }) ; 4. Abstract Flag Logic ■ ** Encapsulate flag checks:** export class DashboardComponent { private readonly showNewDashboard = this . playFeature . isEnabled ("feat-new-dashboard" , false) ; ngOnInit ( ) { this . loadDashboard ( ) ; } private loadDashboard ( ) { if (this . showNewDashboard) { this . loadNewDashboard ( ) ; } else { this . loadLegacyDashboard ( ) ; } } } Configuration Management ■ Environment-specific Configuration ■ // config/play.feature.config.json { "provider" : "launchdarkly" , "clientKey" : "sdk-12345-abcde" , "bootstrap" : { "feat-new-dashboard-2025-q3" : false , "enable-beta-analytics" : true } } User Context ■ // Set user for targeted flags this . playFeature . setUser ({ key : "user-123" , email :

"user@example.com" , name : "John Doe" , custom : { plan : "premium" , region : "us-west" , } , })
; Testing ■ Testing with Different Flag States ■ ** Test with different flag states:** describe (
"DashboardComponent" , ( ) => { it ("should show new dashboard when flag is enabled" , ( ) => {
// Arrange playFeatureService . setFlag ("feat-new-dashboard-2025-q3" , true) ; // Act component
. ngOnInit ( ) ; // Assert expect (component . showNewDashboard) . toBe (true) ; }) ; it ("should
show legacy dashboard when flag is disabled" , ( ) => { // Arrange playFeatureService . setFlag (
"feat-new-dashboard-2025-q3" , false) ; // Act component . ngOnInit ( ) ; // Assert expect (
component . showNewDashboard) . toBe (false) ; }) ; }) ; Testing Checklist ■ Test with flag
enabled Test with flag disabled Test with flag not defined (uses default) Test flag changes during
runtime Test route protection with guard Test directive conditional rendering Test pipe usage in
templates Test user context changes Test offline/fallback behavior Monitoring & Observability ■
Flag Usage Tracking ■ // Log flag usage for analytics this . playFeature . flag$ (
"feat-new-dashboard" , false) . pipe (takeUntil (this . destroy$)) . subscribe ((enabled) => {
this . analytics . track ("feature_flag_viewed" , { flag : "feat-new-dashboard" , enabled , user : this .
currentUser . id , }) ; }) ; Flag Performance Monitoring ■ // Monitor flag check performance const
start = performance . now ( ) ; const isEnabled = this . playFeature . isEnabled (
"feat-new-dashboard" , false) ; const duration = performance . now ( ) - start ; if (duration > 10) {
this . playlog . warn ("Slow feature flag check" , { flag : "feat-new-dashboard" , duration , context :
"dashboard-load" , }) ; } Enforcement ■ ESLint Rules ■ The following ESLint rules enforce Play+
feature flag patterns: No direct playfeature access - Forces use of PlayFeatureService No magic
string flag names - Requires flags to be defined in config No inline flag logic - Encourages
abstraction Code Review Checklist ■ Uses PlayFeatureService instead of direct playfeature
access Provides explicit default values Uses reactive patterns where appropriate Abstracts flag
logic from templates Follows naming conventions Includes cleanup plan for new flags Tests cover
different flag states Migration Guide ■ From Direct playfeature Access ■ Before: import {
playfeature } from "../../../system/play.feature" ; if (playfeature . isEnabled ("my-flag")) { // logic }
After: import { PlayFeatureService } from '../services/playfeature.service' ; constructor (private
playFeature : PlayFeatureService) { } if (this . playFeature . isEnabled ('my-flag' , false)) { //
logic } From Manual Flag Management ■ Before: // Manual flag management private flags = new
Map < string , boolean > ( ) ; setFlag (key : string , value : boolean) { this . flags . set (key , value
) ; } isEnabled (key : string) : boolean { return this . flags . get (key) || false ; } After: // Use
PlayFeatureService constructor (private playFeature : PlayFeatureService) { } setFlag (key :
string , value : boolean) { this . playFeature . setFlag (key , value) ; } isEnabled (key : string ,
defaultValue : boolean = false) : boolean { return this . playFeature . isEnabled (key ,
defaultValue) ; } Troubleshooting ■ Common Issues ■ Flag not working - Check if flag is defined
in config Service not initialized - Ensure PlayFeatureService.initialize() is called Route protection
not working - Verify guard is properly configured Template not updating - Use reactive
observables instead of one-time checks Debug Mode ■ // Enable debug logging this . playFeature
. setFlag ("debug-mode" , true) ; // Check all current flags console . log ("All flags:" , this .
playFeature . getAllFlags ( )) ; // Check flag metadata const metadata = this . playFeature .

getFlagMetadata ("my-flag") ; console . log ("Flag metadata:" , metadata) ; Governance & Hygiene ■ As flag usage grows, discipline is key. Consider establishing formal governance rules: Flag Types : Categorize flags to clarify their purpose (e.g., release-toggle, experiment, ops-switch, kill-switch). Environments : Ensure your provider supports toggling flags independently across different environments (dev, staging, prod). Auditing : Your provider should have an audit log to track who created, toggled, or removed a flag. Lifecycle Metadata : Use tags or descriptions in your provider's UI to add context like owner: team-core-ui , jira: PROJ-123 , introduced: 2025-Q3 . Treat feature flags as first-class citizens in your architecture. Without good hygiene, they become technical debt. Compliance Notes ■ Feature flags must be documented with purpose and cleanup plan All flag names must follow naming conventions Flag changes must be logged for audit purposes User targeting must respect privacy regulations Flag performance must be monitored Unused flags must be cleaned up regularly Developer Checklist ■ Does my new flag have a descriptive name and a cleanup plan? Have I provided a safe defaultValue for my isEnabled check? Is the flag check abstracted (e.g., in a variable) rather than being inline in the UI logic? Have I added the flag to the bootstrap config for offline testing? If the flag is long-term, have I discussed its purpose with the team?